

计算机学院专业必修课

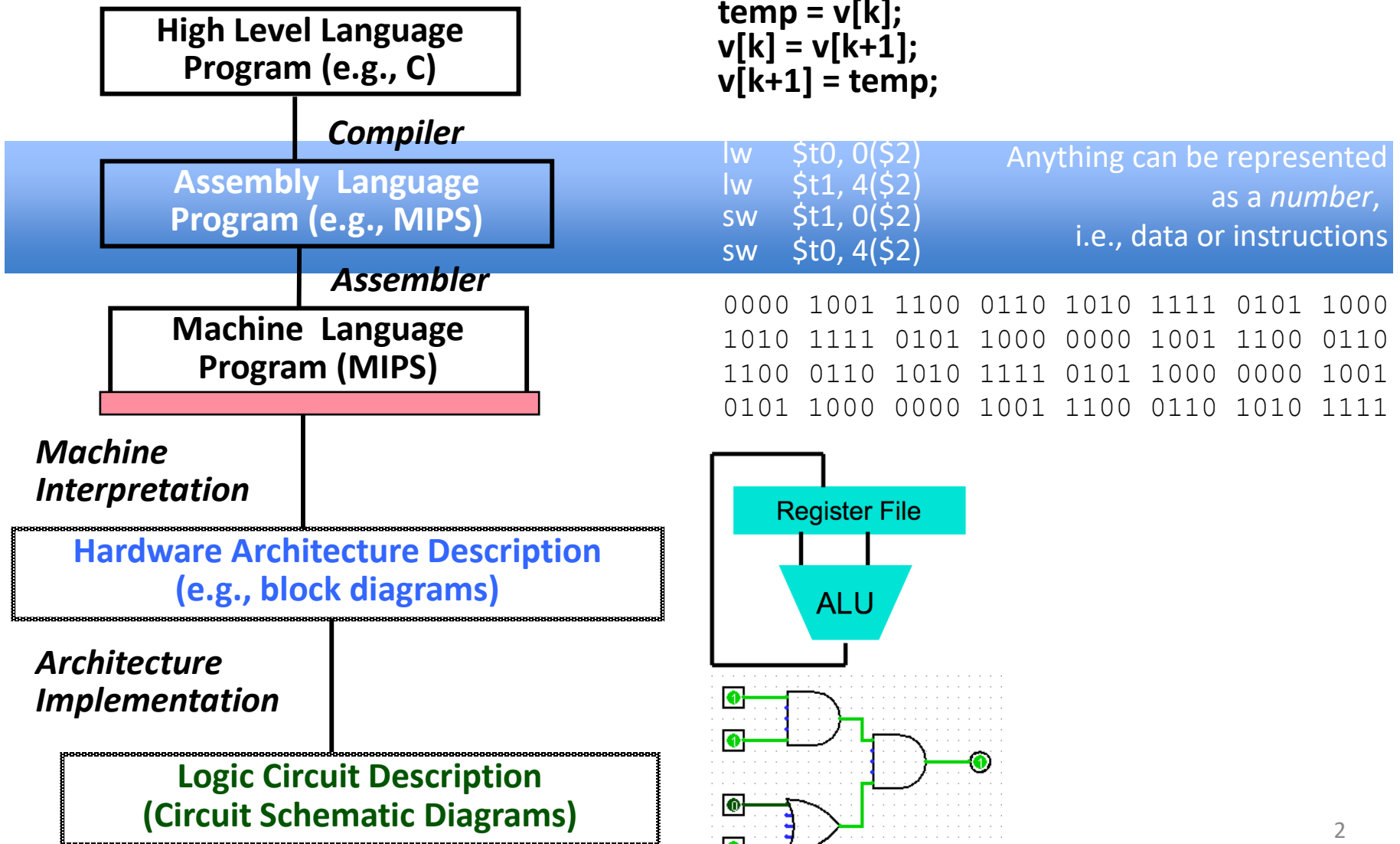
计算机组成

机器语言(1)

高小鹏

北京航空航天大学计算机学院
系统结构研究所

Levels of Representation/Interpretation



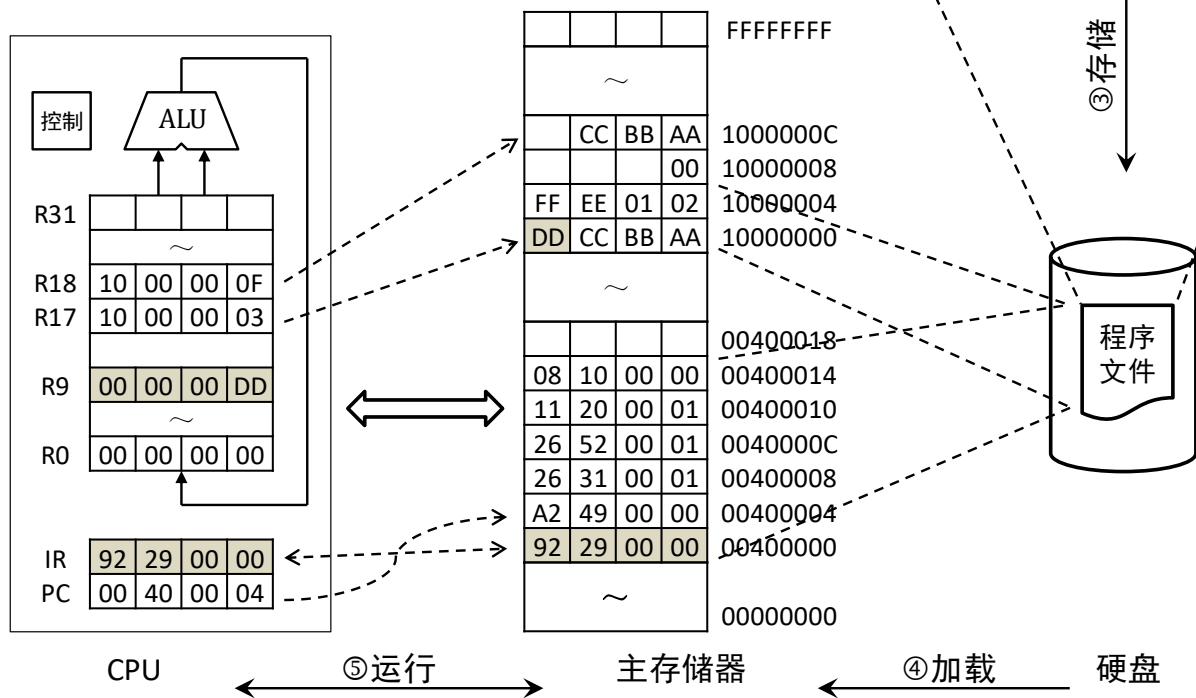
提纲

- 内容主要取材：CS61C的5讲
 - <http://inst.eecs.berkeley.edu/~cs61c/su12>
- 机器语言概述
- 寄存器
- 指令和立即数
- 数据传输指令
- 判断指令

汇编概览

❑ 代码生成；代码保存；代码运行

<pre> while (1) if (*p) { *q = *p ; p++ ; q++ ; } else break ; </pre>		<pre> Loop_Start: lbu \$t1, 0(\$s1) sb \$t1, 0(\$s2) addiu \$s1, \$s1, 1 addiu \$s2, \$s2, 1 beq \$t1, \$0, Loop_End j Loop_Start Loop_End: </pre>		<pre> 10010000001111100000000000000000 0x903E0000 10100010010010010000000000000000 0xA2490000 00100110001100010000000000000001 0x26310001 00100110010100100000000000000001 0x26520001 00010001001000000000000000000001 0x11200001 00001000000100000000000000000000 0x08100000 </pre>
---	--	---	--	--



机器语言^{1/2}

- 指令：CPU理解的“单词”
- 指令集：CPU理解的全部“单词”集合
- Q1：为什么人们有时希望相同的指令集？
 - ◆ 例如：iPhone与iPad使用相同的指令集（ARM）
- Q2：为什么人们有时希望不同的指令集？
 - ◆ 例如：iPhone与Macbook使用不同的指令集（前者是ARM，后者是X86）



- 如果只有一种ISA
 - ◆ 可以很好的利用公共软件，如编译器、操作系统等
- 如果有多种ISA
 - ◆ 针对不同的应用可以选择更适用的ISA
 - ◆ 不同的指令集有不同的设计平衡性考虑
 - 功能、性能、存储器、功耗、复杂度。。。
 - ◆ 会激发竞争和创新

为什么要学习汇编？

- 在更深层次理解计算机行为
 - ◆ 学习如何写更紧凑和有效的代码
 - ◆ 某些情况下，手工编码的优化水平比编译器高
- 对于资源紧张的应用，可能只适合手工汇编
 - ◆ 例如：分布式传感器应用
 - 为了降低功耗和芯片大小，甚至没有OS和编译器



RISC

- 指令集设计早期阶段倾向于：应用有什么操作模式，就增加对应的指令。这导致了 *Complex Instruction Set Computing* (CISC)。另一种对立的设计哲学： *Reduced Instruction Set Computing* (RISC)
 - ◆ 自然界存在2-8定律。程序也类似，为什么？
- RISC的指导思想
 - ◆ 1) 加速大概率事件
 - ◆ 2) 简单就意味着容易设计得更快
 - ◆ 3) 复杂的功能交给软件处理
 - 隐含的物理背景：复杂的功能也是小概率的



RISC设计原则

- ❑ 核心指导思想：CPU越简单，性能越高
- ❑ RISC聚焦在减少指令的数量和复杂度
- ❑ RISC的基本策略
 - ◆ 指令定长：所有指令长度都是1个字（32位）
降低了从存储器中读取指令的复杂度
 - ◆ 简化指令寻址模式：以基地址+偏移为主
降低了从主存中读取操作数的复杂度
 - ◆ ISA的指令不仅数量少，而且简单
降低了指令执行的复杂度
 - ◆ 只有load与store两类指令能够访存
例如，不允许寄存器+存储器或存储器+存储器
 - ◆ 把复杂度留给编译

编译器将高层语言复杂的语句转换为若干个简单的汇编指令



主流的ISA

- Intel 80x86
 - ◆ PC、服务器、笔记本
- ARM (Advanced RISC Machine)
 - ◆ 手机、平板
 - ◆ 出货量最大的RISC：是x86的20倍
- PowerPC
 - ◆ IBM/Motorola/Apple联盟的产物
 - ◆ 航空电子设备：飞控、机载雷达等
 - ◆ 网络设备：交换机、路由器
 - ◆ 引擎控制器



为什么选择MIPS

- 真实：工业界实际使用的CPU
 - ◆ 是设计师在实践中多次迭代、反复权衡的产物
 - ◆ 学习标准就是在学习设计师的思考方法与过程
- 简单：结构简单，易于实现
 - ◆ MIPS是RISC的典型代表
- 生态：软件开发环境丰富，易于学习和实践
 - ◆ 多种模拟器、C编译等



提纲

- 内容主要取材：CS61C的5讲
 - <http://inst.eecs.berkeley.edu/~cs61c/su12>
- 机器语言概述
- 寄存器
- 指令和立即数
- 数据传输指令
- 判断指令

计算机硬件的操作数

- C程序：变量的数量仅仅受限于内存容量
 - ◆ 程序员在声明变量上，通常不需要考虑变量的数量
- ISA：有一组数量有限且固定的操作数，称之为寄存器
 - ◆ 寄存器被内置在CPU内部
 - ◆ 寄存器的优势：速度极快（工作速度小于1ns）
 - ◆ 寄存器的劣势：数量少



MIPS的寄存器

思考

MIPS寄存器为什么不是16个，也不是64个？！

□ MIPS寄存器数量：32

- ◆ 每个寄存器的宽度都是32位
- ◆ 寄存器没有类型（即无正负）
 - 根据指令的功能来解读寄存器值的正负

□ 寄存器数量的是设计均衡的体现

- ◆ 均衡的要素：性能与可用性
- ◆ 数量少：结构简单，速度快，能够存储在CPU内的数据少
- ◆ 数量多：结构复杂，速度慢，能够存储在CPU内的数据多

MIPS的寄存器

TIP

使用寄存器名字
会让代码可读性
更好

- ❑ 寄存器编号：0~31
- ❑ 寄存器表示：\$x（x为0~31），即\$0~\$31
- ❑ 寄存器名字
 - ◆ 程序员变量寄存器
 - \$s0-\$s7 ↔ \$16-\$23
 - ◆ 临时变量寄存器
 - \$t0-\$t7 ↔ \$8-\$15
 - \$t8-\$t9 ↔ \$8-\$15



提纲

- 内容主要取材：CS61C的5讲
 - <http://inst.eecs.berkeley.edu/~cs61c/su12>
- 机器语言概述
- 寄存器
- 指令和立即数
- 数据传输指令
- 判断指令

MIPS指令

- 指令的**一般性**语法格式：1个操作符，3个操作数

op dst, src1, src2

- ♦ op: 指令的基本功能
- ♦ dst: 保存结果的寄存器（“destination”）
- ♦ src1: 第1个操作数（“source 1”）
- ♦ src2: 第2个操作数（“source 2”）



固定的格式有助于使得硬件简单

- ♦ 硬件越简单，延迟就越小，时钟频率就越高



MIPS指令

- ❑ 每条指令只有1个操作
- ❑ 每行写一条指令
- ❑ 很多指令与C运算高度相关
 - ◆ 如：=, +, -, *, /, &, |
- ❑ 一行C代码会对应多条指令



MIPS指令示例

□ 假设：变量a，b和c分别存储在\$s1，\$s2和\$s3

◆ $a \longleftrightarrow \$s1$ ， $b \longleftrightarrow \$s2$ ， $c \longleftrightarrow \$s3$

□ 整数加法指令

◆ C: $a = b + c$

◆ MIPS: `add $s1, $s2, $s3`



□ 整数减法指令

◆ C: $a = b - c$

◆ MIPS: `sub $s1, $s2, $s3`

MIPS指令示例

- 假设: $x \leftrightarrow \$s0$, $a \leftrightarrow \$s1$, $b \leftrightarrow \$s2$, $c \leftrightarrow \$s3$, $d \leftrightarrow \$s4$
- C语句: $x = (a + b) - (c + d);$
- MIPS汇编程序片段

执行序 ↓	1	add <u>$\\$t1$</u> , $\$s3$, $\$s4$	# $t1 = c+d$
	2	add <u>$\\$t2$</u> , $\$s1$, $\$s2$	# $t2 = a+b$
	3	sub $\$s0$, <u>$\\$t2$</u> , <u>$\\$t1$</u>	# $a = (a+b) - (c+d)$
			
		MIPS汇编程序	注释

- ◆ $\$t1$, $\$t2$: 临时变量寄存器

- ◆ 注释: 提高可读性; 帮助追踪寄存器/变量的分配与使用

- #: 是注释语句的开始

0号寄存器

- 由于0在程序中出现频度极高，为此MIPS为0设置了专属寄存器
 - ◆ 表示方法：\$0或\$zero
 - ◆ 值恒为0：读出的值恒为0；写入的值被丢弃
 - 指令的dst为\$0：指令使用本身无错，但执行时没有实际意义

□ 示例

假设： $a \longleftrightarrow \$s1$, $b \longleftrightarrow \$s2$, $c \longleftrightarrow \$s3$, $d \longleftrightarrow \$s4$

```
1  add $s3, $0, $0      # c=0
2  add $s1, $s2, $0      # a=b
```

立即数

Q
为什么没有subi?

- 指令中出现的**常量数值**被称为立即数
- 语法格式

op dst, src, imm

- 立即数替代了第2个操作数

- 示例

假设: $a \longleftrightarrow \$s1$, $b \longleftrightarrow \$s2$, $c \longleftrightarrow \$s3$, $d \longleftrightarrow \$s4$

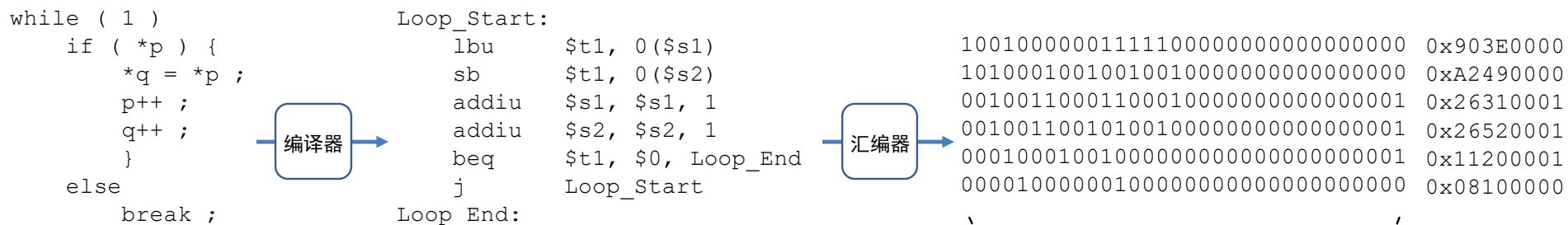
```
1  addi $s1, $s2, 5      # a=b+5
2  addi $s3, $s3, 1      # c++
```

提纲

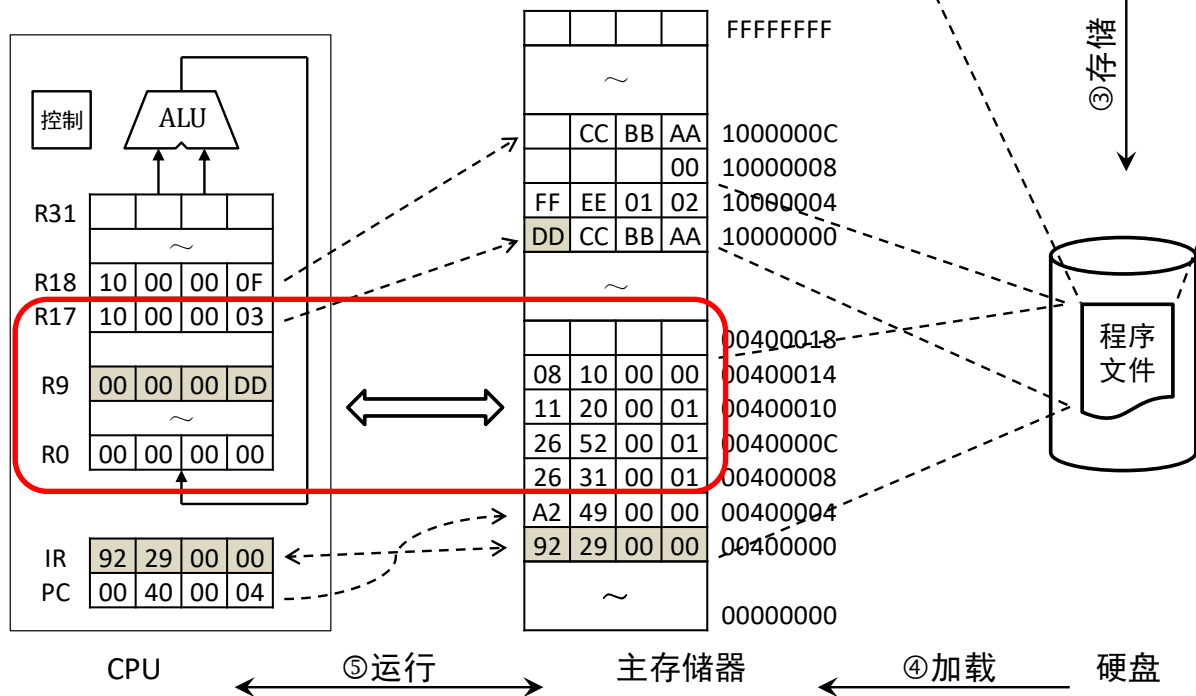
- 内容主要取材：CS61C的5讲
 - <http://inst.eecs.berkeley.edu/~cs61c/su12>
- 机器语言概述
- 寄存器
- 指令和立即数
- 数据传输指令
- 判断指令

数据传输概览

□ 数据传输：寄存器与存储器之间的数据交换



C程序 → ①编译 → MIPS汇编程序 → ②汇编 → 指令的机器码（2进制格式和16进制格式）



数据传输

- 虽然C中的变量可以映射为寄存器，但面临挑战：有限的寄存器个数无法满足无限的变量需求
 - ◆ 1、变量的个数无限：一个函数的变量可以是无限多的
 - ◆ 2、变量的容量巨大：如数组这样的大型数据结构
- 解决问题途径：主存
- 由于MIPS只能对寄存器与立即数进行运算，因此必须有特定的数据传输指令实现主存单元与寄存器的数据交换
 - ◆ LOAD类指令：寄存器 \leftarrow 主存单元
 - ◆ STORE类指令：寄存器 \rightarrow 主存单元

数据传输

□ 语法格式

op reg, off(base)

- ◆ reg: 写入或读出的寄存器
- ◆ base: 存储基地址的寄存器
 - 该寄存器的作用就是指针
 - 由于是地址，因此base的值被作为无符号数
- ◆ off: 以字节为单位的偏移
 - off是立即数，可正可负

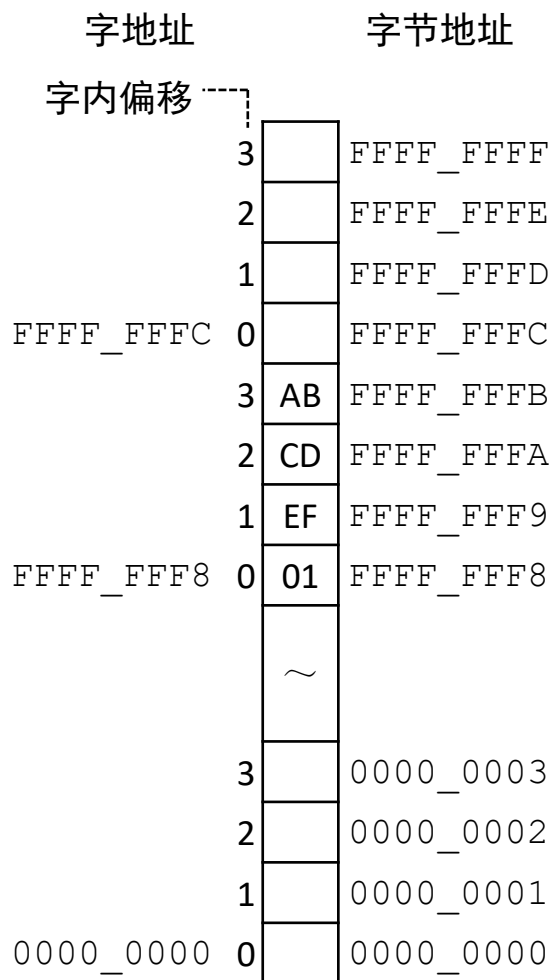
□ 读写的存储单元的实际地址=base+off

□ 这种寻址方式被称为：基地址+偏移

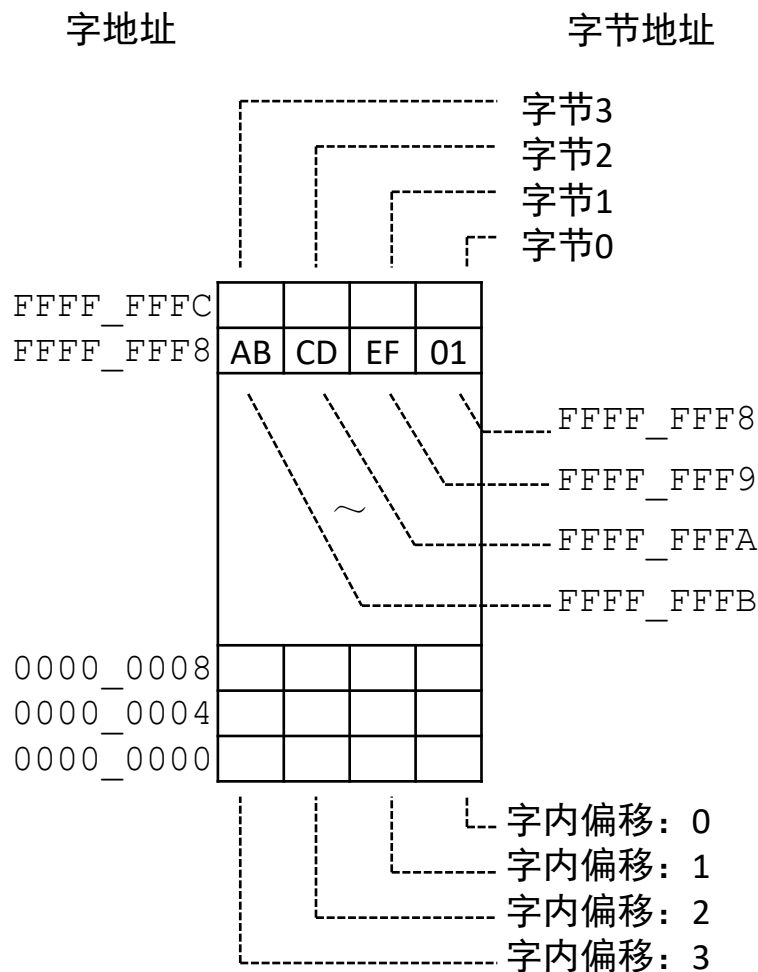
□ 该寻址方式可以表示任意某个存储单元的地址

存储空间的视图

- 字节是存储器地址的基本单位；字地址与字内最左字节地址相同



A. 以字节为单元的存储视图



B. 以字为单元的存储视图

数据传输指令

- ❑ 加载字：lw (Load Word)
 - ◆ 从地址为 $\text{base}+\text{off}$ 的存储单元中读出字，然后写入 reg
- ❑ 存储字：sw (Store Word)
 - ◆ 读取 reg 中的字，写入地址为 $\text{base}+\text{off}$ 的存储单元
- ❑ 示例

假设： $A[] \longleftrightarrow \$s3$, $a \longleftrightarrow \$s0$

1	lw	$\$t0, 12(\$s3)$	# $\$t0 = A[3]$
2	add	$\$t0, \$s2, \$t0$	# $\$t0 = A[3] + a$
3	sw	$\$t0, 40(\$s3)$	# $A[10] = A[3] + a$

TIP

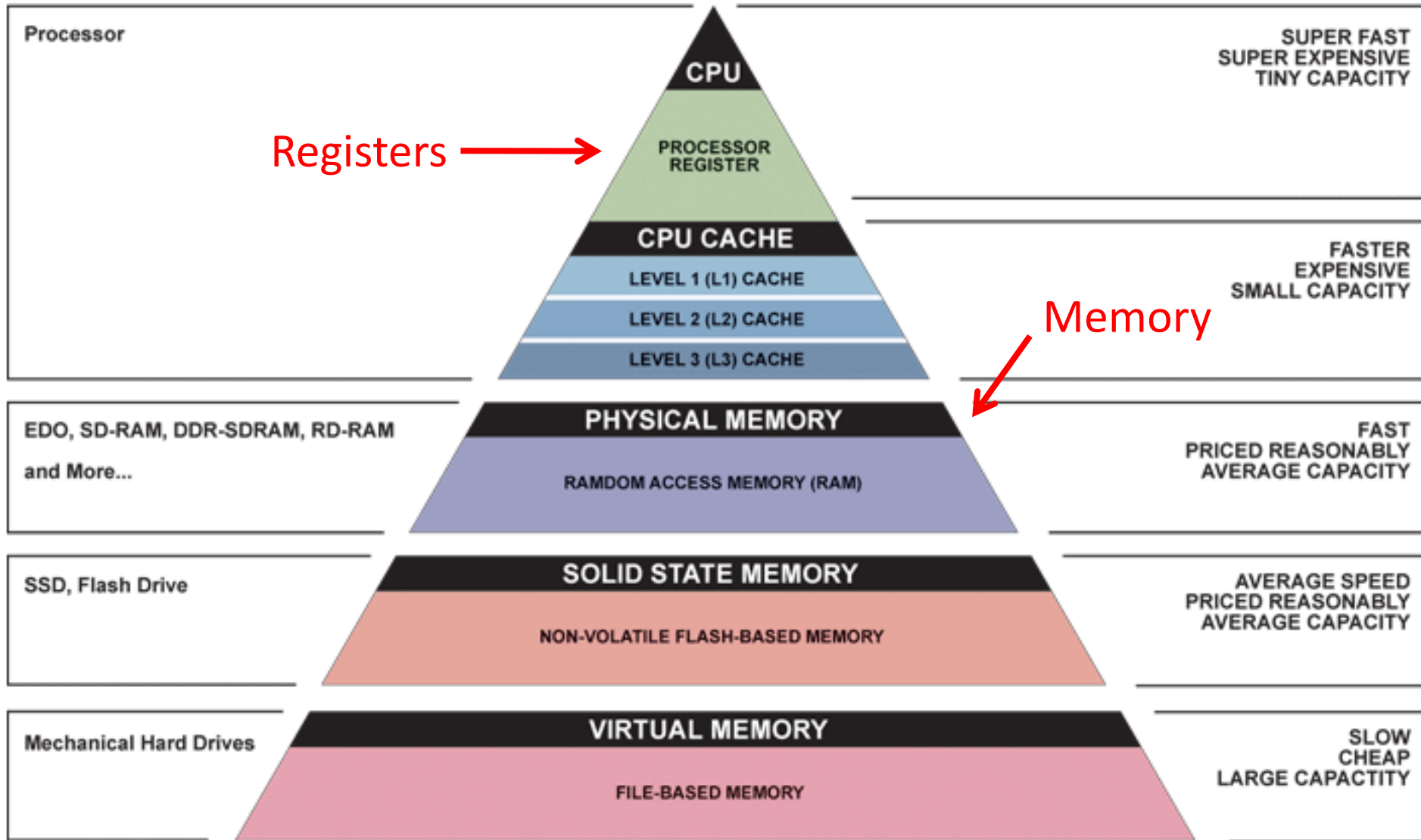
lw/sw偏移必须是4的倍数

寄存器 vs. 存储器

- 变量比寄存器多怎么办？
 - ◆ 把最常用的变量保存在寄存器中
 - ◆ 其他不常用的保存在存储器中
- 为什么不把变量都放在存储器中？
 - ◆ 寄存器比存储器快100~500倍
 - ◆ 用寄存器可以设计更灵活的微结构，例如寄存器堆（后续介绍）
 - 寄存器堆可以同时读2个操作数，并写入1个操作数



Great Idea #3: Principle of Locality/ Memory Hierarchy



lb的用法

- 专用的加载/存储字节的指令

```
lb $s0, 0($s1)
```

```
sb $s0, 1($s1)
```

TIP

lb/sb偏移可以不是4的倍数

- lb: 需要进行24位的符号扩展

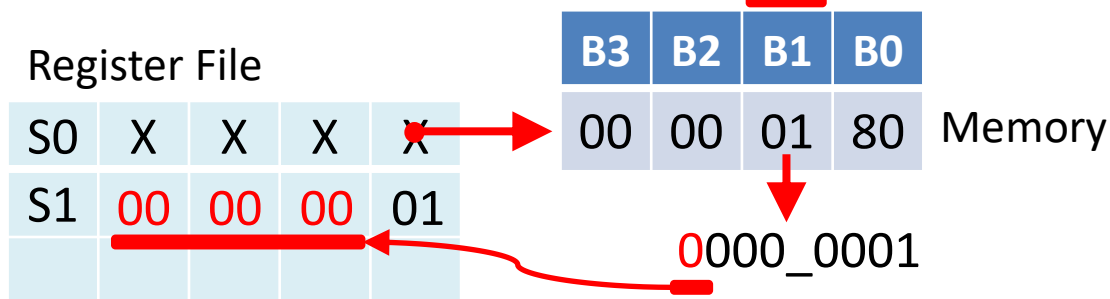
Q

为什么sw没有符号扩展?

- 读入字节的最高位被视为符号位；向高24位进行符号扩展

- 示例: 假设 $*(\$s0) = 0x00000180$

```
lb $s1, 1($s0) # $s1 = 0x00000001
```



lb的用法

- 专用的加载/存储字节的指令

```
lb $s0, 0($s1)
```

```
sb $s0, 1($s1)
```

TIP

lb/sb偏移可以不是4的倍数

- lb: 需要进行24位的符号扩展

Q

为什么sw没有符号扩展?

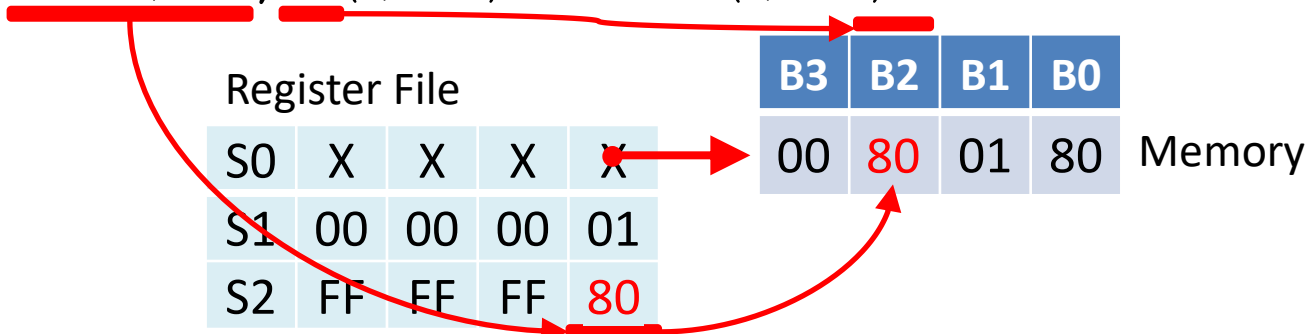
- 读入字节的最高位被视为符号位；向高24位进行符号扩展

- 示例：假设 $*(\$s0) = 0x00000180$

```
lb $s1, 1($s0) # $s1=0x00000001
```

```
lb $s2, 0($s0) # $s2=0xFFFFFFFF80
```

```
sb $s2, 2($s0) # *($s0)=0x00800180
```



加载和存储指令汇总

- 字操作：偏移必须是4的倍数
 - ◆ lw、sw
- 半字操作：偏移必须是2的倍数
 - ◆ lh、lhu
 - ◆ sh
- 字节操作
 - ◆ lb、lbu
 - ◆ sb

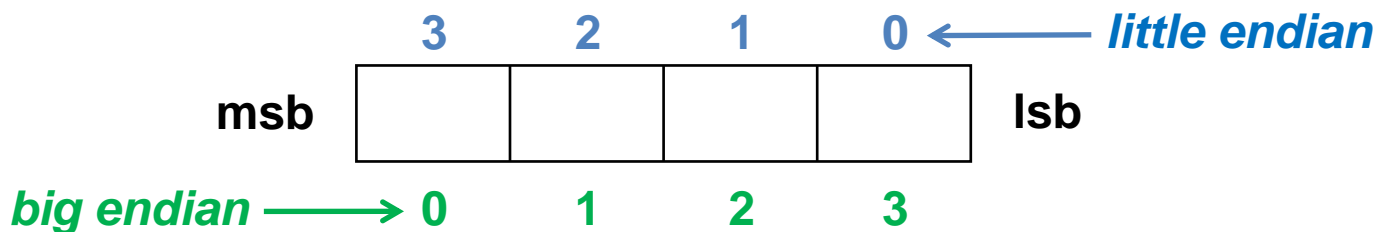
TIP

lbu/lhu：没有符号扩展



大小印第安

- 大印第安：最高有效字节在字内的最低地址
- 小印第安：最高有效字节在字内的最高地址



- MIPS：同时支持2种类型
 - 本课程用小印第安

提纲

- 内容主要取材：CS61C的5讲
 - <http://inst.eecs.berkeley.edu/~cs61c/su12>
- 机器语言概述
- 寄存器
- 指令和立即数
- 数据传输指令
- 判断指令

基本的决策机制

- C: 有if-else, for, while, do-while等语句块
 - ◆ 决策机制: 根据条件转移并执行相应的语句块
- MIPS: 通过标号机制来实现转移
 - ◆ MIPS没有语句块的概念, 只有地址的概念
 - ◆ 每条指令都对应一个word地址
 - ◆ 为了提高可读性, 汇编程序使用标号来标记其后的指令的地址
 - 标号是由字符串+':'组成。例如: **ForBegin:**
 - ◆ 汇编语言再通过跳转机制跳转到标号处, 从而实现转移

TIP

C也有类似的机制, 例如goto, 但被认为是不好的编程风格

决策指令

□ Branch If Equal (beq): 相等时转移

- ◆ `beq reg1, reg2, label`
- ◆ 如果`reg1`的值=`reg2`的值, 则转移至`label`处执行

□ Branch If Not Equal (bne): 不等时转移

- ◆ `bne reg1, reg2, label`
- ◆ 如果`reg1`的值 \neq `reg2`的值, 则转移至`label`处执行

□ Jump (j): 无条件转移

- ◆ `j label`
- ◆ 无条件转移至`label`处执行
- ◆ 对应C的goto机制

用beq构造if-else

- 与C不同之处: beq/bne构造if-else是条件为TRUE则转移!

C Code:

```
if (i==j) {  
    a = b /* then */  
} else {  
    a = -b /* else */  
}
```

In English:

- 如果TRUE, 执行THEN语句块
- 如果FALSE, 执行ELSE语句块

MIPS (beq):

```
# i → $s0, j → $s1
```

```
# a → $s2, b → $s3
```

```
beq $s0, $s1, ???
```

```
??? ← 可以去除该标号
```

```
sub $s2, $0, $s3
```

```
j end
```

```
then:
```

```
add $s2, $s3, $0
```

```
end:
```

用bne构造if-else

- 与C不同之处: beq/bne构造if-else是条件为TRUE则转移!

C Code:

```
if (i==j) {  
    a = b /* then */  
} else {  
    a = -b /* else */  
}
```

In English:

- 如果TRUE, 执行THEN语句块
- 如果FALSE, 执行ELSE语句块

MIPS (bne):

```
# i → $s0, j → $s1  
# a → $s2, b → $s3  
  
bne $s0, $s1, ???  
???  
add $s2, $s3, $0  
j    end  
  
else:  
sub $s2, $0, $s3  
  
end:
```

循环

- C语言有3种循环：for, while, do...while
 - ◆ 3种语句是等价的，即任意一种循环都可以改写为其他两种循环
- MIPS只需要一种决策机制即可
 - ◆ 核心：根据条件转移



实战：从C到MIPS

- 实例：字符串赋值

- C代码

```
/* Copy string from p to q */  
char *p, *q;  
while ( (*q++ = *p++) != '\0' ) ;
```

- Q: 代码结构有什么特征?

- ◆ 单一的while循环
- ◆ 退出循环是一个相等测试

实战：从C到MIPS

▣ 实例：字符串赋值

▣ C代码

```
/* Copy string from p to q */  
char *p, *q;  
while ( (*q++ = *p++) != '\0' ) ;
```

▣ 代码结构有什么特征？

- ◆ 单一的while循环
- ◆ 退出循环是一个相等测试

Q: 2种写法的区别在哪里？

```
while ( *p )  
    *q++ = *p++ ;
```

实战：从C到MIPS

▣ STEP1: 构造循环的框架

```
# copy String p to q
```

```
# p→$s0, q→$s1 (pointers)
```

```
Loop:
```

```
# $t0 = *p
```

```
# *q = $t0
```

```
# p = p + 1
```

```
# q = q + 1
```

```
# if *p==0, go to Exit
```

```
# go to Loop
```

```
    j Loop
```

```
Exit:
```

实战：从C到MIPS

▣ STEP2：构造循环主体

copy String p to q

$p \rightarrow \$s0$, $q \rightarrow \$s1$ (pointers)

```
Loop: lb    $t0, 0($s0)    # $t0 = *p
      sb    $t0, 0($s1)    # *q = $t0
      addi  $s0, $s0, 1    # p++
      addi  $s1, $s1, 1    # q++
      beq   $t0, $0, Exit  # if *p==0, go to Exit
      j     Loop           # go to Loop
```

Exit:

Q

1个字符需要6条指令。能否优化？

实战：从C到MIPS

□ 优化代码（减少了1条指令）

copy String p to q

$p \rightarrow \$s0$, $q \rightarrow \$s1$ (pointers)

```
Loop: lb    $t0, 0($s0)    # $t0 = *p
      sb    $t0, 0($s1)    # *q = $t0
      addi  $s0, $s0, 1    # p = p + 1
      addi  $s1, $s1, 1    # q = q + 1
      bne   $t0, $0, Loop  # if *p != 0, go to Loop
```



乘除法指令

- ❑ 乘除法指令计算结果：不是直接写入32个通用寄存器，而是保存在2个特殊寄存器HI与LO
- ❑ 用2条专用指令读写HI/LO
 - “move from HI” (`mfhi dst`)
 - “move from LO” (`mflo dst`)
- ❑ **Multiplication** (`mult`)
 - ◆ `mult src1,src2`
 - ◆ `src1*src2`: LO保存结果的低32位，HI保存结果的高32位
- ❑ **Division** (`div`)
 - ◆ `div src1,src2`
 - ◆ `src1/src2`: LO保存商，HI保存余数



乘除法指令

▣ 示例：用div求模

mod using div: $\$s2 = \$s0 \bmod \$s1$

mod:

div $\$s0, \$s1$ # LO = $\$s0 / \$s1$

mfhi $\$s2$ # HI = $\$s0 \bmod \$s1$

算术溢出

- 复习：当计算结果的位数超出计算机硬件的实际能保存的位数，即为**溢出**
 - ◆ 换言之，即没有足够的位数保存结果
- MIPS会检测溢出（并且溢出发生时**产生错误**）
 - ◆ 有unsigned关键字的算术类指令忽略溢出

Overflow Detection	No Overflow Detection
<code>add dst, src1, src2</code>	<code>addu dst, src1, src2</code>
<code>addi dst, src1, src2</code>	<code>addiu dst, src1, src2</code>
<code>sub dst, src1, src2</code>	<code>subu dst, src1, src2</code>

算术溢出

□ 示例

复习：这是最小的负数！

\$s0=0x80000000, \$s1=0x1
add \$t0,\$s0,\$s0 # overflow (error)
addu \$t1,\$s0,\$s0 # \$t1=0
addi \$t2,\$s0,-1 # overflow (error)
addiu \$t3,\$s0,-1 # \$t3=0x7FFFFFFF
sub \$t4,\$s0,\$s1 # overflow (error)
subu \$t5,\$s0,\$s1 # \$t5=0x7FFFFFFF

位运算指令

□ 假设: $a \rightarrow \$s1, b \rightarrow \$s2, c \rightarrow \$s3$

Instruction	C	MIPS
And	$a = b \ \& \ c;$	<code>and \$s1, \$s2, \$s3</code>
And Immediate	$a = b \ \& \ 0x1;$	<code>andi \$s1, \$s2, 0x1</code>
Or	$a = b \ \ c;$	<code>or \$s1, \$s2, \$s3</code>
Or Immediate	$a = b \ \ 0x5;$	<code>ori \$s1, \$s2, 0x5</code>
Not Or	$a = \sim(b \ \ c);$	<code>nor \$s1, \$s2, \$s3</code>
Exclusive Or	$a = b \ ^ \ c;$	<code>xor \$s1, \$s2, \$s3</code>
Exclusive Or Immediate	$a = b \ ^ \ 0xF;$	<code>xori \$s1, \$s2, 0xF</code>

移位指令

- C语言有移位操作，MIPS也定义了多条移位指令
- 移位指令可以从3个维度来分析
 - ◆ 方向：左移还是向右移
 - ◆ 性质：逻辑移位还是算术移位
 - 对于向左移位来说，低位永远是补0
 - 只有向右移位，才存在高位是补0还是符号位的选择问题。如果补0，那就是逻辑移位，如果是补符号位，则为算术移位。
 - ◆ 移位量：对于32位寄存器，移动位数的合理最大取值为31，即0x1F
 - 如何在指令中表示这个移位量呢？
 - 方式1：由一个5位的立即数来表示移位量
 - 方式2：用某寄存器的值来表示移位量。如果用寄存器来表示移位量，则只有该寄存器的最低5位被CPU识别为移位量，而高27位无论取何值均无意义。

Q
根据上述3个维度，应该定义几条指令？



移位指令

■ MIPS共6条移位指令

- ◆ 如果使用立即数：只有0~31有效
- ◆ 如果使用寄存器：寄存器的低5位有效（按5位无符号数对待）

指令	功能	示例
sll	逻辑左移	sll \$t0, \$s0, 16
srl	逻辑右移	srl \$t0, \$s0, 16
sra	算术右移	sra \$t0, \$s0, 16
sllv	逻辑可变左移	sllv \$t0, \$s0, \$s1
srlv	逻辑可变右移	srlv \$t0, \$s0, \$s1
srav	算术可变右移	srav \$t0, \$s0, \$s1

移位指令

■ 示例

```
addi $t0,$0,-256 # $t0=0xFFFFFFFF00
sll $s0,$t0,3 # $s0=0xFFFFF800
srl $s1,$t0,8 # $s1=0x0FFFFFFF
sra $s2,$t0,8 # $s2=0xFFFFFFFF

addi $t1,$0,-22 # $t1=0xFFFFFEEA
# low 5: 0b01010
sllv $s3,$t0,$t1 # $s3=0xFFFC0000
# same as sll $s3,$t0,10
```

作业

- 《计算机组成与设计》
- WORD: 2.1、2.4、2.5、2.6
- MARS
 - 2.5.2